Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 1045–1054

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

PFFTC: An improved fast Fourier transform for the IBM Cell Broadband Engine

Andrew Shaffer[a,b,*], Bruce Einfalt[a], Padma Raghavan[b]

[a]*Applied Research Laboratory, The Pennsylvania State University, State College, PA 16801, USA*
[b]*Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA*

Abstract

The Fast Fourier Transform (FFT) is a widely used algorithm that is frequently employed in environments where high performance is critical. In the context of embedded systems, FFTs often have hard runtime constraints and must be evaluated using limited hardware. In this paper, we present a partitioned FFT algorithm (PFFTC) for the Cell Broadband Engine (Cell BE) that improves upon previous FFT implementations for this platform. PFFTC has three main phases to (i) partition the problem into independent sub-problems, (ii) solve the sub-problems in parallel, and (iii) combine the results of the sub-problems to obtain the solution to the original problem. PFFTC includes optimizations for exploiting data transfer parallelism, avoiding unnecessary communication through careful data routing, avoiding data dependency stalls with instruction-level double buffering, and minimizing synchronization overhead through the use of an "asynchronous" signal-based barrier. We evaluate the performance of PFFTC and other FFT algorithms for the Cell BE. Our results indicate that PFFTC attains a peak processing rate of 33.6 GFLOPS, and achieves speedups ranging from 31% to 62% over the fastest previous Cell BE FFT algorithm's reported performance for complex single-precision FFTs with 1,024-16,384 data points.

Keywords: Fast Fourier Transform; Cell Broadband Engine

1. Introduction

The Fast Fourier Transform (FFT) is a widely used algorithm that is frequently employed in environments where high performance is critical. In the context of embedded systems, FFTs often have hard runtime constraints and must be evaluated using limited hardware. FFTs that are on the critical path of a larger algorithm must be evaluated quickly to avoid delaying the completion of the entire algorithm. These factors indicate the importance of developing FFT algorithms that can solve individual FFT problems with very low latency.

In this paper, we present Partitioned Fastest Fourier Transform for the IBM Cell Broadband Engine (PFFTC) – an FFT algorithm for the Cell Broadband Engine (Cell BE) that improves upon previous FFT implementations for this platform. PFFTC has three main phases to (i) partition the problem into independent sub-problems, (ii) solve the sub-problems in parallel, and (iii) combine the results of the sub-problems to obtain the solution to the original

* Corresponding author. Tel.: 1-814-321-3202 .
E-mail address: aps148@psu.edu .

1877-0509
doi:10.1016/j.procs.2010.04.116

problem. PFFTC includes optimizations for exploiting data transfer parallelism, avoiding unnecessary communication through careful data routing, avoiding data dependency stalls with instruction-level double buffering, and minimizing synchronization overhead through the use of an "asynchronous" signal-based barrier.

We evaluate the performance of PFFTC and other FFT algorithms for the Cell BE. Our results indicate that PFFTC attains a peak processing rate of 33.6 GFLOPS, and achieves speedups ranging from 31% to 62% over the fastest previous Cell BE FFT algorithm's reported performance for complex single-precision FFTs with 1,024-16,384 data points [1]. We compute that our algorithm achieves performance ranging from 15% to 30% of the peak FFT performance that is possible on the Cell BE for these problem sizes.

The remainder of this paper is organized as follows: we first describe the Cell BE architecture and prior FFT implementations for the Cell BE in Section 2. We describe the design of our new PFFTC algorithm and discuss its optimization features in Section 3. We then describe our testing environment and methodology in Section 4, and present the results of our performance and accuracy evaluations in Section 5. We conclude with a review of our optimization techniques that can be applied to other problems and a discussion of future work in Section 6.

2. Background

The Cell BE is a multicore architecture containing heterogeneous processing elements. It is capable of very high single-precision floating point computation rates, and has been extensively detailed in previous literature [2,3,4,5]. In this section, we provide a brief overview of the Cell BE architecture and several prior FFT implementations that have been developed for this platform.

The heart of the Cell BE's design is an array of up to eight Synergistic Processing Elements (SPEs), which are specialized SIMD processors optimized for high single-precision floating point performance. Each SPE has access to a personalized local store and a large quadword register file. The SPEs are augmented by a general-purpose PowerPC Processing Element (PPE), which is well suited to coordinate the activities of the SPEs and to support traditional programs like the Linux operating system. The SPEs, the PPE, and the system's main memory are connected by a high-bandwidth Element Interconnect Bus (EIB), which provides communication between each pair of processing elements and between each processing element and the system's main memory at the rate of 25.6 GB/sec. The Cell BE architecture provides application developers with low-level control over the system's processing elements and over most data movement across the EIB, and is capable of sustaining a peak single-precision performance exceeding 204.8 GFLOPS.

Previous FFT implementations for the Cell BE have taken a variety of approaches. FFTW operates by dynamically generating an execution plan built from a library of statically optimized FFT algorithm components [6,7]. The selection of components for the execution plan is based upon the size of the problem being solved and the performance of each of the components on the underlying hardware platform. This makes FFTW highly portable, but does not allow it to take advantage of optimizations that are specific to the Cell BE.

In contrast, Mercury Computer Systems' FFT_ZIPX algorithm relies on a highly tuned FFT kernel that has been extensively modified to run efficiently on a single SPE [8]. The FFT_ZIPX kernel attains nearly the theoretical limit of a single SPE's FFT performance [9], but the Cell BE implementation of this kernel cannot be run on any other system because it is so specialized. Also, because FFT_ZIPX is designed to solve an FFT problem using only a single SPE, it leaves most of the Cell BE's hardware unused. Multiple instances of FFT_ZIPX can be run simultaneously to utilize the Cell BE's SPEs to the maximum extent possible when solving large sets of FFT problems, but the implementation's use of only one SPE per FFT problem means that it cannot achieve the theoretical minimum latency for solving a single FFT problem on the Cell BE.

A third approach is used by Bader and Agarwal's FFTC algorithm, in which a single FFT problem is distributed across all eight SPEs in the Cell BE [1]. The FFTC algorithm uses an iterative out-of-place approach that distributes each butterfly stage across all eight SPEs, with the data for each stage being taken from main memory, processed, and then returned to main memory at the end of the stage. This design necessitates moving the problem data between main memory and the SPEs' local stores multiple times throughout the problem execution. It also requires synchronizing the SPEs at the end of each butterfly stage. However, FFTC's approach allows all of the SPEs to be used in the solution of a single FFT problem. The FFTC package reports the best performance for solving small single-precision complex FFT problems on the Cell BE as of the time of its publication, and it still has the best previously published performance for these problems of any Cell BE FFT algorithm of which we are aware.

3. PFFTC: A Partitioned FFT Algorithm for the IBM Cell BE

PFFTC is a low-latency FFT implementation that solves single-precision complex FFT problems up to 16,384 data points using 2-8 SPEs. It requires no working buffer in main memory, allows prefetching of upcoming problem data to improve throughput, and supports efficient on-the-fly switching between forward and inverse FFTs.

As shown in Figure 1a, PFFTC has three main phases to (i) partition the problem into independent sub-problems, (ii) solve the sub-problems in parallel, and (iii) combine the results of the sub-problems to obtain the solution to the original problem. In this section, we describe the operations performed by PFFTC during each phase, and also the optimizations that we used to make the algorithm operate efficiently.

3.1. Partitioning Stage

The partitioning stage is designed to divide a single FFT problem into four, eight or sixteen smaller independent sub-problems that can be solved independently in parallel. The sub-problems that are created by the partitioning stage correspond exactly to the sub-problems that would be considered by the lowest level of recursion in a recursive implementation of the FFT algorithm if the recursive algorithm were cut off at a fixed recursion depth of two, three, or four levels of in-place recursive partitioning. However, to increase efficiency, the partitioning is performed in a single pass over the data by means of hard-coded permutation functions that scan the input from start to finish. These functions apply the composition of either two, three or four levels of recursive partitioning in a single operation.

The partitioning stage is only ever executed on SPE 0 and SPE 1, regardless of how many SPEs are allocated to PFFTC. SPE 0 partitions the problem's real data and SPE 1 partitions the problem's imaginary data. The use of two SPEs to perform the partitioning stage reduces the stage's processing time by 50%, and also ensures that the full 25.6 GB/sec bandwidth of the main memory is used to transfer the problem data to the SPEs. The use of only two SPEs to perform the partitioning stage also ensures that the partitioned data for each sub-problem is located in exactly two contiguous blocks in the local stores of the SPEs, simplifying data routing and distribution in the subsequent solution stage.

Double buffering is used to bring the problem data into the local stores of SPE 0 and SPE 1 in multiple pieces, allowing partitioning to begin before the transfer of problem data has been completed. Two data blocks are used to minimize DMA control overhead for problems with 2,048 or fewer data points. For larger problem sizes, four data blocks are retrieved with the double buffering.

Once the partitioning functions have finished their processing, all of the sub-problems' real data resides in contiguous blocks on SPE 0 and all of the sub-problems' imaginary data resides in contiguous blocks on SPE 1. SPE 0 and SPE 1 finish the stage by forwarding the first sub-problem that will be solved by each SPE to the SPEs that will solve the sub-problems.

3.2. Solution Stage

During the solution stage, the independent sub-problems created during the partitioning stage are solved in parallel on the allocated SPEs. The sub-problems are distributed among the SPEs in round-robin fashion to balance the workload, and are solved using an iterative FFT kernel derived from the FFTC algorithm.

To avoid memory access stalls in the solution stage FFT kernel, we use a technique similar to double-buffering at the individual instruction level, as shown in Figure 1b. Specifically, we use one set of variables to hold the initial data for each butterfly computation, a second set of variables to hold the intermediate values, and a third set of variables to hold the final output of the butterfly computation until it can be written back to the local store. The first iteration of the butterfly stage loop is manually unrolled so that the twiddle factors needed by the loop and the input data needed for the second loop iteration can be loaded concurrently with the execution of the first loop iteration. Then, for each successive iteration, the input values for the subsequent iteration are prefetched and the output values of the previous iteration are output while the intermediate and final values for the current iteration are computed. The output of the final loop iteration is written out to the local store after the loop has ended. This approach enables us to avoid memory access stalls for all but the very first accesses performed by each loop iteration, maintaining high performance.

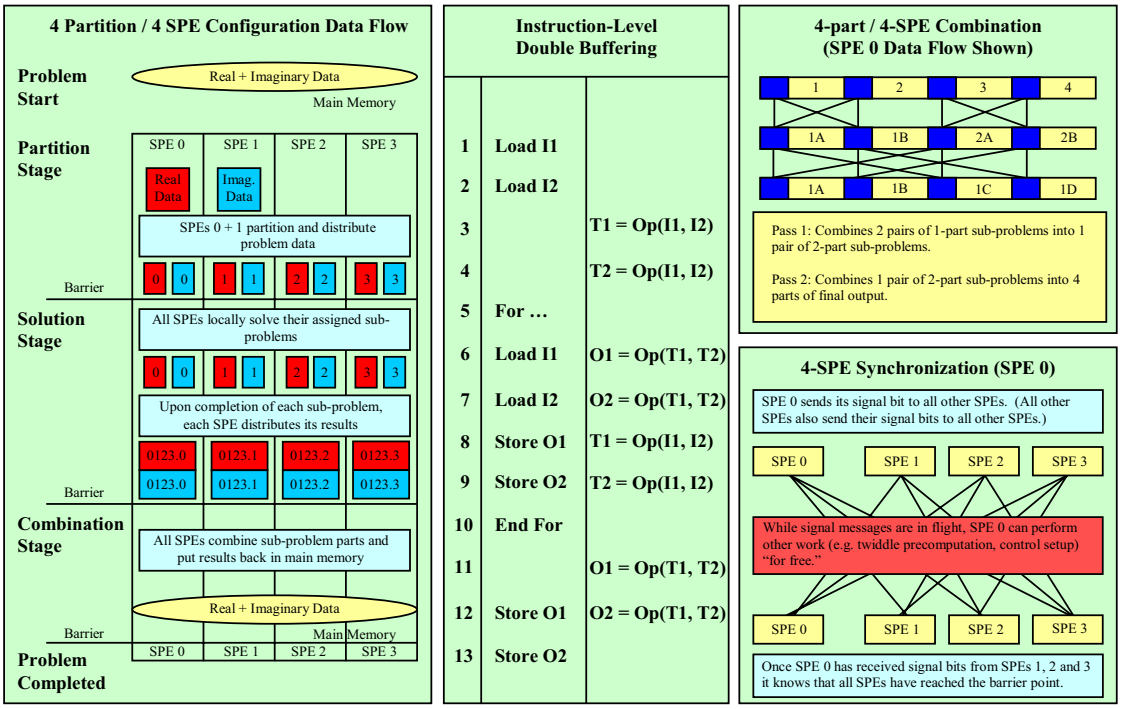

Fig. 1. (a - left) PFFTC Data Flow Diagram; (b – center) Instruction-Level Double Buffering; (c – top right) Communication Free Combination Stage; (d – bottom right) Asynchronous Synchronization

Additionally, we further improve performance by manually unrolling the pipelined butterfly stage loop body four times. This degree of unrolling is possible because of the large size of the SPE register file. The execution of four butterfly computations per loop iteration improves performance by providing four sets of independent instructions in the loop body that can be ordered to avoid all data dependency stalls between the operations producing the intermediate results and the operations needing these values as input to compute the final results for each butterfly computation. In combination with the instruction-level double buffering optimization discussed previously, this loop unrolling allows our algorithm to avoid data dependency and local store access stalls altogether during its main processing loop.

To keep the solution stage running as quickly as possible, all of the sub-problems solved on each SPE except for the forwarded first sub-problem are retrieved from the local stores of SPE 0 and SPE 1 using double buffering. The combined bandwidth of SPE 0 and SPE 1 is 51.2 GB/sec, making the solution stage decidedly compute-bound. We take advantage of the excess communication capacity available while the solution stage is running by having the solution stage forward the output of each sub-problem to the SPEs where it will be needed in the subsequent combination stage as soon as each sub-problem has been solved.

3.3. Combination Stage

The combination stage corresponds directly to an iterative version of the recursive combinations performed by a recursive implementation of the FFT algorithm. Our algorithm to combine the sub-problems operates in-place, and performs as many passes over the data as the recursion depth cut-off that is used when creating the sub-problems in the partitioning stage.

As shown in Figure 1c, each butterfly stage in an in-place recursive FFT algorithm combines pairs of adjacent sub-problems into a single larger sub-problem with size equal to the combined size of the original sub-problems, and

each butterfly stage computation places its output values into the same memory locations as its input values. Thus, if one SPE is provided with the first data point of each sub-problem, it will be able to compute the butterfly computations involving the first data point of each sub-problem for an initial butterfly stage. Then, because the memory locations of the output for each butterfly stage are the same as the butterfly computation's input data, it will be able to compute the butterfly computations involving the first data point of the sub-problems from the previous butterfly stage's output, and also the butterfly computations involving the midpoint of the sub-problems from the previous butterfly stage's output. This pattern continues for all points in the original set of sub-problems and for as many levels of the recursion as were created during the partitioning stage, so if an SPE is provided with the m^{th} through n^{th} elements of each sub-problem, it can compute all of the remaining butterfly stages corresponding to these data points without requiring any additional data from any other SPE.

We maximize DMA efficiency and balance the load of the combination stage across all allocated SPEs by assigning the first portion of each sub-problem to SPE 0, the second portion to SPE 1, and so on for all allocated SPEs, such that the portions are balanced evenly. The portion of each sub-problem that is needed by each SPE for the combination stage is forwarded to where it is needed as each sub-problem is computed during the solution stage. Then, once the solution stage has finished, all of the data is already in place for the combination stage to run to completion without any further inter-SPE communication.

Lastly, the combination stage is designed to ensure that the FFT result is returned to main memory as quickly as possible. This is accomplished by beginning the transfer of each portion of the problem result back to main memory as soon as it has been computed.

3.4. Asynchronous Synchronization

At the end of each of the three major stages in the PFFTC algorithm it is necessary to synchronize the SPEs. After the partitioning stage, a synchronization is necessary to ensure that the forwarded first sub-problem that will be solved by each SPE has been completely transmitted before the SPE begins to solve it. This synchronization also ensures that the data for any later sub-problems is ready to be prefetched before it is retrieved for use in the solution stage. Then, once the solution stage has been completed, a second synchronization is needed to ensure that all data forwarding performed during the solution stage has been completed before allowing the SPEs to work with this data in the combination stage. Finally, after the combination stage, a third synchronization is necessary to ensure that every SPE has completely transmitted its portion of the original problem result back to main memory before notifying the PPE that the result is ready.

We chose to implement the synchronization operations using a signal-based method to reduce the impact of these synchronization operations on PFFTC's throughput. Each SPE has two signal registers that can be placed in logical-OR mode to accumulate the set bits of any messages that are received, and we assign each SPE used by our algorithm a unique signal message equal to 2^{spe_id}. As shown in Figure 1d, whenever a synchronization is performed, each SPE sends its unique signal message to every other SPE upon reaching the beginning of the synchronization operation. These signals can be enqueued almost instantaneously and they are sent asynchronously, so the SPE can resume processing that is unrelated to the synchronization operation as soon as the signal enqueuing is complete. Once the SPE has completed its non-synchronized work, it can check which of the other SPEs have reached the synchronization start point by polling its signal register to see which bits have been set. After the bit corresponding to each allocated SPE has been set in the signal register, the polling SPE has established that all of the other SPEs have also reached the synchronization start point, and it can safely continue beyond the synchronization operation.

Although the signal-based barrier requires a significantly more messages to be sent between SPEs than is necessary in a tree-based synchronization, these messages can all be sent independently from one another, allowing the communication on all SPEs to start as early as possible and to be amortized over a longer runtime without affecting performance. For instance, if eight SPEs are allocated to PFFTC, SPEs 2-7 can send all of their synchronization messages for the synchronization at the end of the partitioning stage while SPE 0 and SPE 1 are performing the partitioning stage. Then, when SPE 0 and SPE 1 have finished their work for the partitioning stage, each of these two SPEs needs to send only seven independent signal messages to complete the barrier operation. Since the EIB has four bus channels and all of the signals being sent are independent, sending seven messages from two SPEs will require a maximum of four signal delay periods. Likewise, at the end of the combination stage, every

SPE can send its synchronization signal to every other SPE except SPE 0 even before the problem data has been fully transmitted back to main memory. This is possible because only SPE 0 needs to know when each SPE has completed its data transmission so that it can notify the PPE when the problem has been completed. If eight SPEs are allocated, this allows all but seven of the synchronization signals to be sent early, so the number of signals on the algorithm's critical path is minimal. Since the EIB has four bus channels and all of the signals being sent are independent, sending one message from each of seven SPEs will require a maximum of two signal delay periods. Both of these cases experience significantly less than the six signal delay periods on the algorithm's critical path needed to perform the gather and scatter operations that would be used by a tree-based synchronization method.

4. Environment & Methodology

In this section, we discuss our testing environment and the methodology that we used to evaluate the latency and accuracy of our new PFFTC algorithm and the latency of prior Cell BE FFT packages.

We made use of two PlayStation 3 systems and one IBM QS20 blade server in our evaluations. The PlayStation 3 systems each support a single Cell BE with six SPEs, and the blade server supports two Cell BEs which each contain eight SPEs. All of these platforms had the IBM Cell BE Software Development Kit (SDK) version 2.1 installed.

The FFTW package requires an outdated version of the IBM SDK, so we chose not to test it empirically. Instead, we present its reported results for both the PlayStation 3 and the blade server platforms from the FFTW website [7].

Next, the FFT_ZIPX package from Mercury Computer Systems only runs under Yellow Dog Linux 6.0. The blade server implementation for this package is quite expensive, so we evaluated only the less expensive PlayStation 3 implementation for this package. Our PlayStation 3 for the FFT_ZIPX tests runs under Yellow Dog Linux kernel 2.6.23-9.ydl6.1.

After evaluating FFT_ZIPX, we found that Bader and Agarwal's FFTC requires eight SPEs to operate, so it could only be evaluated on the blade server system [1]. Our blade server system runs under Linux kernel 2.6.23.

Lastly, we designed our PFFTC package to operate using between 2-8 SPEs, so it is capable of operating on both the PlayStation 3 and the blade server platforms. We evaluate its performance on both of these systems. The PlayStation 3 system used to test our PFFTC package runs under Linux kernel 2.6.16.

When measuring the runtime of each algorithm, we track the time that elapses between the first SPE being notified that a problem is ready in main memory until the PPE is notified that the problem result is fully stored in the main memory. We measure the runtime needed to solve 1,000 FFT problems in sequence and then report the average runtime needed to solve each problem in the 1,000 problem test to ensure that our tests run long enough to be timed accurately. We also report the best average timing result from a sample of 100 test runs to filter out the effect of transient system processes and network traffic, and we put each FFT algorithm through a "warm-up" run before beginning its main timing loop to avoid the effects of TLB faults and page faults.

Our test loop for each package is structured to completely solve each FFT problem from start to finish before beginning any work for any subsequent problems so that our timing effectively measures the latency of solving a single FFT problem in isolation. The only exception to this is that we allow the PFFTC package to prefetch the memory address of each FFT problem's input data during the solution of the previous FFT problem in the test series, so that this address is resident in the local store of each SPE at the beginning of the problem. This accommodation for PFFTC is intended to make its timing results more comparable with those of the other packages, because the other packages do not perform the step of retrieving unique problem information for each FFT problem that is solved in their test series. Instead, the other FFT packages transmit information about the location of problem input buffers once before beginning their main timing loop and then solve this same FFT problem repeatedly.

The PFFTC package accepts command line parameters to specify how many partitions should be created and how many SPEs should be allocated. For each problem size, we evaluated every possible combination of partition counts and SPE allocations. We report the best average runtime over all of these combinations as the runtime for each problem size for PFFTC.

Additionally, for the PFFTC algorithm we ran a series of 1,000 tests for each problem size that we considered to verify the correctness of our algorithm. Each test took as input an appropriately sized array of normally-distributed random single-precision complex input data with a mean of zero and a standard deviation of one and performed an FFT identity operation on the test input data by running a forward FFT followed immediately by an inverse FFT on

the output of the original forward FFT problem. The partition and SPE count parameters for each test were chosen to match those that were found to provide the fastest runtime for each problem size. This sequence of operations returns the original input values exactly if the component FFT operations introduce no errors at all, so any differences that occur between the input and output are due to the implementation of the FFT algorithm. We computed the inf-norm of the differences between the output of our FFT identity operation and the original input for each test case, and report the maximum inf-norm observed across the 1,000 tests for each problem size as a measure of the total error introduced by our algorithm's implementation.

5. Results

In this section, we report and analyze the performance and accuracy results of our empirical testing for the PFFTC package and prior Cell BE FFT packages. We also discuss the effect of recursion depth cut-off in the PFFTC partitioning stage and the impact of SPE allocation on overall PFFTC performance.

Using the metric that an FFT problem requires $5*N*\log_2(N)$ floating point operations, our timing results indicate that PFFTC attains a peak processing rate of 33.6 GFLOPS. As shown in Figure 2, PFFTC achieves speedups ranging from 31% to 62% over Bader and Agarwal's FFTC algorithm [1], which had the highest previously reported FFT performance for problems with 1,024-16,384 data points.

Given that at most $5*N*\log_2(N)$ floating point operations are performed for an FFT problem with N data points, the 25.6 GB/sec main memory bandwidth is a critical bottleneck in Cell BE FFT performance for small problem sizes. Although a Cell BE with eight SPEs is capable of 204.8 GFLOPS of single-precision floating point performance, the need for each FFT problem to be moved from the main memory to the SPE local stores and for a result of equal size to be moved back to main memory creates a minimum communication delay for each size of FFT problem that prevents the peak single-precision floating point processing rate from being attained. Even if all FFT computation could be masked with communication operations, the communication requirements alone would lead to the theoretical performance limits shown in Table 1.

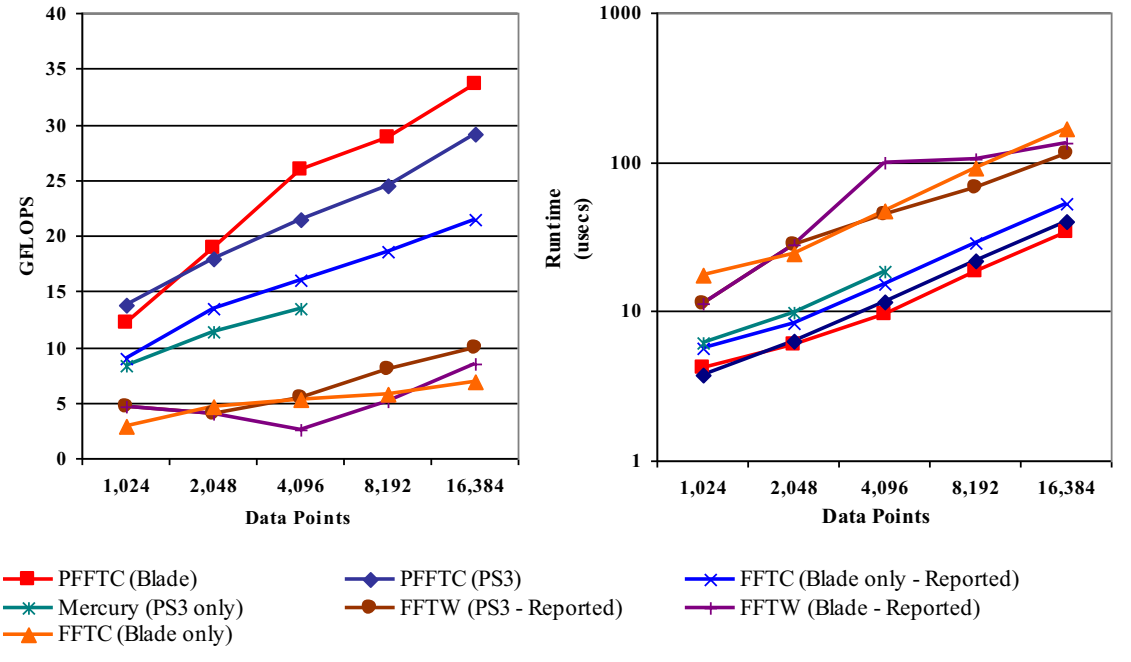

Fig. 2. FFT GFLOPS and Runtime vs. Problem Size

Table 1. Maximum Cell BE FFT Performance for Small Problem Sizes

Problem Size (points)	Data Size (bytes)	2-Way Transfer Time (microseconds)	Floating Point Operations	Maximum GFLOPS
1,024	8,192	0.64	51,200	80
2,048	16,384	1.28	112,640	88
4,096	32,768	2.56	245,760	96
8,192	65,563	5.12	532,480	104
16,384	131,072	10.24	1,146,880	112

In comparison to these maximum theoretical performance values, our PFFTC algorithm attains efficiency ratings of between approximately 15% and 30%, as shown in Table 2.

Table 2. PFFTC Efficiency for Small Problem Sizes

Problem Size (points)	Maximum GFLOPS	PS3 GFLOPS	Blade GFLOPS	PS3 Efficiency	Blade Efficiency
1,024	80	13.8	12.2	17.3%	15.2%
2,048	88	17.9	18.9	20.3%	21.5%
4,096	96	21.5	25.9	22.4%	27.0%
8,192	104	24.5	28.9	23.6%	27.8%
16,384	112	29.1	33.6	25.9%	30.0%

These results reflect the best performance achieved for each problem size across all possible combinations of partition counts and SPE allocations. As shown in Table 3, we find that a variety of workload distribution strategies allow PFFTC to achieve its best performance for different problem size and platform combinations.

Table 3. Optimal Partitioning and SPE Allocation for Small Problem Sizes on PlayStation 3 and Blade Server

Problem Size (points)	Optimal PS3 Partitioning / SPE Allocation	Optimal Blade Partitioning / SPE Allocation
1,024	4 / 4	4 / 4
2,048	4 / 4	8 / 8
4,096	4 / 4	8 / 8
8,192	16 / 6	16 / 8
16,384	16 / 6	16 / 8

For smaller problem sizes, communication has a more significant impact on runtime than computation. Thus, we find that PFFTC attains its best performance for these problems using workload distribution strategies that reduce the number of required communication and synchronization messages at the expense of sacrificing some computational power. On the other hand, computation plays the most significant role in the runtime of the larger problem sizes. For these problem sizes, PFFTC attains its best performance by allocating the maximum possible number of SPEs.

For all but the largest of the problem sizes that we considered, PFFTC achieves its best performance when the number of partitions is equal to the number of allocated SPEs. This strategy negates any benefits from double buffering throughout the algorithm, but reduces the total number of messages that must be sent across the bus throughout the algorithm, which is a critical bottleneck for small problems. For the larger problems, sending only a few large sub-problems across the bus would require the algorithm to wait for longer periods while the large sub-problems are distributed among the SPEs at the end of each stage. Also, there is more total computation time required for these problems, so there is a good opportunity to effectively mask some of the sub-problem communication with computation through double buffering. For these problems, we find that PFFTC attains its best performance by maximizing the number of sub-problems and distributing them across the maximum possible number of SPEs.

We note that although the blade server achieves its best performance using more than four SPEs for the 2,048 and 4,096 point problem sizes, the PlayStation 3 system still attains its best performance using only four SPEs for these problem sizes. This occurs because the PlayStation 3 has a maximum of only six SPEs available. If all six SPEs are allocated for these problem sizes the additional SPEs will add communication cost to the overall runtime because

they will have to participate in each synchronization operation, but the full benefit of their computational power will not be applied to the problem because neither four, eight or sixteen sub-problems can be distributed evenly across six SPEs.

PFFTC has significantly higher performance than any prior Cell BE FFT package that we evaluated. Bader and Agarwal's FFTC algorithm reported the next best performance [1], although we were not able to reproduce these results during our testing. After FFTC, the Mercury FFT_ZIPX package achieved nearly the same performance as FFTC's reported results while using only a single SPE, but this package can only solve problem sizes up to 4,096 data points [8]. After the FFT_ZIPX package, the FFTW package had the slowest performance, which was not unexpected given FFTW's emphasis on portability instead of maximum platform-specific performance [6,7]. We also found FFTC to attain performance similar to FFTW's reported results in our empirical testing. When we contacted the FFTC authors to investigate the performance of their algorithm, they indicated that their reported results were obtained on a pre-production blade server that is no longer available.

With respect to accuracy, we found that PFFTC's implementation solves FFT problems of all the sizes that we considered with only a small amount of error. A summary of the maximum inf-norm results obtained in our testing is shown in Table 4.

Table 4. PFFTC Accuracy Results

Problem Size (points)	Maximum Inf-norm Observed on PS3	Maximum Inf-norm Observed on Blade
1,024	0.000066	0.000066
2,048	0.000185	0.000075
4,096	0.000380	0.000164
8,192	0.000168	0.000170
16,384	0.000399	0.000373

We believe that the error reported above occurs mainly because the Cell BE SPEs implement non-compliant floating point arithmetic that is less accurate even than the standard single-precision floating point arithmetic that is available on most traditional computing platforms [4]. The exact amount of error reported for each problem size is dependent upon the partitioning and SPE count parameters that are used for each problem size, as these parameters affect the exact values computed for the twiddle factors in the solution stage and combination stage of the PFFTC algorithm, and also the numbers of butterfly stages computed using each of these sets of twiddle factors. However, we believe that the effect of the partitioning and SPE count parameters is small when compared to the total error due to basic floating point rounding errors. Altogether, the PFFTC algorithm reports total error that is very small when compared to the magnitude of its input values and the number of floating point operations that are being performed.

6. Conclusions and Future Work

In summary, we present PFFTC, our high performance FFT implementation for the Cell BE. PFFTC has three main phases to (i) partition the problem into independent sub-problems, (ii) solve the sub-problems in parallel, and (iii) combine the results of the sub-problems to obtain the solution to the original problem. PFFTC includes optimizations for exploiting data transfer parallelism, avoiding unnecessary communication through careful data routing, avoiding data dependency stalls with instruction-level double buffering, and minimizing synchronization overhead through the use of an "asynchronous" signal-based barrier method. These optimization techniques can generally be applied to the optimization of any complex problem that is solved on the Cell BE.

Our PFFTC algorithm currently attains a peak performance of 33.6 GFLOPS for a problem size of 16,384 data points, and attains speedups ranging from 31% to 62% over the fastest previous Cell BE FFT algorithm's reported performance for FFTs with 1,024-16,384 data points [1]. PFFTC thus attains the lowest latency solution of any Cell BE FFT package of which we are currently aware. It also achieves this performance with high accuracy, which is limited only by the precision of the non-IEEE-754 compliant floating point unit present in the Cell BE's SPEs.

PFFTC could be extended to support double-precision FFTs in the future, which would improve its usefulness for solving scientific computing problems requiring higher floating point precision. However, the Cell BE is capable of only a fraction of its peak single-precision floating point performance when performing double-precision operations,

and the utilization of double-precision data would significantly increase the unmasked communication time required during the partitioning and combination stages of the PFFTC algorithm. Also, using double-precision data would require allocating larger amounts of the SPE local store for twiddle factors and thus reduce the maximum problem size that PFFTC could solve. Because it would be necessary to re-write nearly all of PFFTC's hand-optimized code to support a double-precision implementation, and because such an implementation is not expected to achieve outstanding performance, we have decided to defer a double-precision implementation of PFFTC until we encounter a real-world problem that requires such an FFT implementation on the Cell BE platform.

Overall, we note that the Cell BE faces a significant memory bottleneck when solving FFT problems in isolation from other operations. This bottleneck limits the Cell BE to just over 54% of its theoretical peak floating point performance for the largest FFT problem sizes that we considered, even assuming that the data transfer for solving multiple FFT problems in sequence can be arranged to fully utilize the main memory bandwidth and that the computation time for solving the FFTs can be reduced to balance exactly with the data transfer time for each problem. If the full potential of the Cell BE is to be realized for complex applications, it will be necessary to find ways to perform more useful computational work on the problem data once it has been brought from main memory to the SPEs than what would be possible by using the PFFTC algorithm as a library call to complete a single step in a larger application.

Acknowledgements

This research was supported in part by the National Science Foundation through grants CCF-0830679, CNS-0720749 and OCI-0821527. The PFFTC source code may be provided for research purposes upon request to the authors.

References

1. Bader, D. and Agarwal, V. "FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine." *College of Computing, Georgia Institute of Technology*, 2007. [Online]. Available: http://www.cc.gatech.edu/~bader/papers/FFTC-HiPC2007.pdf. [Accessed Feb. 25, 2009].
2. Buttari, A., Luszczek, P., Kurzak, J., Dongarra, J., and Bosilca, G. "A Rough Guide to Scientific Computing on the PlayStation 3." *Innovative Computing Laboratory, University of Tennessee Knoxville*, Tech. Rep. UT-CS-07-595, 2007. [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/PAPERS/scop3.pdf. [Accessed: Feb. 25, 2009].
3. Chen, T., Raghavan, R., Dale, J., and Iwata, E. "Cell Broadband Engine Architecture and its First Implementation," *IBM Corporation*, 2005. [Online]. Available: http://www.ibm.com/developerworks/power/library/pa-cellperf. [Accessed: Feb. 25, 2009].
4. IBM Corporation Technical Staff, *Cell Broadband Engine Programming Handbook v. 1.1*, IBM Corporation, 2007.
5. IBM Corporation, "developerWorks: Cell Broadband Engine Resource Center," *IBM Corporation*. [Online]. Available: http://www.ibm.com/developerworks/power/cell. [Accessed: Feb. 25, 2009].
6. Frigo, M. and Johnson, S., "The Design and Implementation of FFTW3," *Proceedings of the IEEE 93 (2), 216-231 (2005). Invited paper, Special Issue on Program Generation, Optimization and Platform Adaptation*. [Online]. Available: http://www.fftw.org/fftw-paper-ieee.pdf. [Accessed: Feb 25, 2009].
7. FFTW.org, "Fastest Fourier Transform in the West," *FFTW.org*. [Online]. Available: http://www.fftw.org/cell. [Accessed: Feb. 25, 2009].
8. Mercury Computer Systems, Inc., "Algorithm Performance on the Cell Broadband Engine Processor, Revision 1.1.2," *Mercury Computer Systems, Inc.*, 2006. [Online]. Available: http://www.mc.com/uploadedFiles/Cell-Perf-Simple.pdf. [Accessed: Feb. 25, 2009].
9. Cico, L., Cooper, R., and Greene, J., "Performance and Programmability of the IBM/Sony/Toshiba Cell Broadband Engine Processor," Mercury Computer Systems, Inc., 2006. [Online]. Available: http://www.mc.com/uploadedFiles/CellPerfAndProg-3Nov06.pdf. [Accessed: Feb. 25, 2009].